UNIVERSITY OF COPENHAGEN

Precomputing method lookup

Jul, Eric

Publication date:
2009

Document version
Peer reviewed version

Citation for published version (APA):
Jul, E. (2009). *Precomputing method lookup*. Paper presented at 3rd International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, Paphos, Cyprus. http://icooolps.loria.fr/icooolps2008/Papers/ICOOOLPS2008_paper09_Jul_final.pdf

Download date: 04. May. 2023

Precomputing Method Lookup
Eric Jul
ICOOOLPS 2008

Abstract. This paper looks at Method Lookup and discusses the Emerald approach to making Method Lookup more efficient by precomputing method lookup as early as possible — even moving it back into the compilation phase, if possible, thus eliminating method lookup entirely for many simple procedure calls.

1 Introduction

1.1 Original Motivation

Smalltalk has been a very influential language. However, some of its features designed for flexibility, were also hard to implement efficiently. We believe that at least some of Smalltalk's performance problems are caused by the absence of static typing: If only the compiler had more information available to it about the set of operations that can be invoked on an object, it could surely optimize the process of finding the right code, i.e., performing *method lookup*. This inspired the designers of the programming language Emerald [1] to design a mechanism for making Method Lookup more efficient. In the following, we describe this mechanism which was successful in eliminating most Method Lookup in Emerald. Subsequent advances such as inline caches have largely eliminated the "lookup penalty".

1.2 Abstract and Concrete Types

From our experience with Eden, we knew that a distributed system was never complete: it was always open to extension by new applications and new objects. Today, in the era of the Internet, the fact that the world is "under construction" has become a cliché, but in the early 1980s the idea that all systems should be extensible — we called it the "open world assumption" — was new.
A consequence of this assumption is that an Emerald program needed to be able to operate on objects that do not exist at the time that the program was written, and, more significantly, on objects whose *type* is not known when the application was written. How could this be? Clearly, an application must have *some* expectations about the operations that could be invoked on a new object, otherwise the application could not hope to use the object at all. If an existing program P had minimal expectations of a newly injected object, such as requiring only that the new object accept the *run* invocation, many objects would satisfy those expectations. In contrast, if another program Q required that the new object understand a larger set of operations, such as *redisplay*, *resize*, *move* and *iconify*, fewer objects would be suitable.

We derived most of Emerald's type system from the open world assumption. We coined the term *concrete type* to describe the set of operations understood by an actual, concrete object, and the term *abstract type* to describe the declared type of a piece of programming language syntax, such as an expression or an identifier. The basic question that the type system attempted to answer was whether or not a given object (characterized by a concrete type) supported enough operations to be used in a particular context (characterized by an abstract type). Whenever an object was bound to an identifier, which could happen when any of the various forms of assignment or parameter binding were used, we required that the concrete type of the object *conform* to the abstract type declared for the identifier. In essence, conformity ensured that the concrete type was "bigger" than the abstract type, that is, the object understood a superset of the required operations, and that the types of the parameters and results of its operations also conformed appropriately.

Basing Emerald's type system on conformity distinguished it from contemporary systems such as CLU, Russell, Modula-2, and Euclid, all of which required equality of types. It also distinguished Emerald's type system from systems in languages like Simula that were based on subclassing, that is, on the ancestry of the object's implementation. In a distributed system, the important questions are not about the implementation of an object (which is what the subclassing relation captures) but about the operations that it implements.

1.3 Type Checking and Binding of Types

Another consequence of the open world assumption was that sometimes type checking had to be performed at run time, for the very simple reason that neither the object to be invoked nor the code that created it existed until after the invoker was compiled. This requirement was familiar to us from our experience with the Eden Programming Language [2]. However, Eden used completely different type systems (and data models) for those objects that could be created dynamically and those that were known at compile time.

For Emerald, we wanted to use a single consistent object model and type system. Herein lies an apparent contradiction. By definition, compile-time type checking is done at compile time, and an implementation of a typed language should be able to guarantee at compile time that no type errors will occur. However, there are situations where an application must insist on deferring type checking, typically because an object with which it wishes to communicate will not be available until run time.

Our solution to this dilemma provided for the consistent application of conformity checking at either compile time or run time. If enough was known about an object at compile time to guarantee that its type conformed to that required by its context, the compiler certified the usage to be type-correct. If not enough was known, the type-check was deferred to run time. In order to obtain useful diagnostics, we made the design decision that such a deferral would occur only if the programmer requested it explicitly, which was done using the **view**...**as** primitive, which was partially inspired by qualification in Simula 67 [3,4].

Consider the example

```
var unknownFile: File
    ...
r ← (view unknownFile as Directory).Lookup["README "]
```

Without the **view**...**as** *Directory* clause, the compiler would have indicated a type error, because *unknownFile*, as a *File*, would not understand the *Lookup* operation. With the clause, the compiler treated *unknownFile* as a *Directory* object, which would understand *Lookup*. In consequence, **view** ... **as** required a dynamic check that the type of the object bound to *unknownFile* did indeed conform to *Directory*. Thus, successfully type-checking an Emerald program at compile time did not imply that no type errors would occur at run time; instead it guaranteed that any type errors that *did* occur at run time would do so at a place where the programmer had explicitly requested a dynamic type check.
The **view**...**as** primitive later appeared in C++.
Note that **view**...**as** is similar to casting in Java in that the interface view changes. However, in terms of type system and implementation, there is a substantial difference: Java does not check the types when casting but merely that the class of the casted object *implements* the specified interface (or an interface that inherets the specified interface). In Emerald, the *implements* relationship is not defined by a syntatic construct but rather implicitly by conformity: *Any* object that has the operations specified by the interface (Abstract Type) *implements* the interface.
Partially inspired by the **inspect** statement of Simula 67 [3,4], we also introduced a Boolean operator that returned the result of a type check. This allowed a programmer to check for conformity before attempting a **view**...**as**.

1.4 Operation Invocation

A performance problem plaguing object systems, e.g., Smalltalk, that were contemporary with Emerald was the cost of finding the code to execute when an operation was invoked on an object. This process was then generally known by the name "method lookup"; indeed it still is, in Emerald it is called operation invocation. In Smalltalk, method lookup involved searching method dictionaries starting at the class of the target object and continuing up the inheritance class hierarchy until the code was located. We thought that if Emerald didn't do static type checking, each operation invocation would require searching for an implementation of an operation with the correct name, which would be expensive — although, because we did not provide inheritance, not as expensive as in Smalltalk. In a language like Simula in which each expression had a static type that uniquely identified its *implementation*, each legal message could be assigned a small integer and these integers could be used as indices into a table of pointers to the code of the various methods. In this way, Simula was able to use table lookup rather than search to find a method (and C++ still does so). We though that static typing would give Emerald

the same advantage, and this was one of the motivations for Emerald's static type system.
However, even with static typing, there is still a problem in Emerald: except for the above-mentioned primitive types, knowing the type of an identifier at compile time tells us *nothing* about the implementation of the object to which it will be bound at run time. This is true even if the program submitted to the compiler contains only a single implementation that conforms to the declared type, because it is always possible for another implementation to arrive over the network from some other compiler. Thus, the Emerald implementation would still have to search for the appropriate method: the advantage that static typing would give us would be a guarantee that such a method existed.

1.5 More Efficient Method Lookup using The AbCon Mechanism

The most dynamic form of Method Lookup is to, for a given invocation, search the concrete type for the operation that is to be invoked. Consider the example from above:

var *unknownFile: File*
...
$r \leftarrow$ (**view** *unknownFile* **as** *Directory*).*Lookup*["*README* "]
result $\leftarrow$ *r.Lookup*(*something*)

When invoking the method *Lookup* file object assigned to *r*, the implementation can merely access the object, find its reference to its own concrete type, and then search the concrete type for the method.
The Emerald implementation used several techniques to avoid this expensive dynamic search process.
First, it is often the case that *dataflow analysis* can be used to ascertain that an object has a specific concrete type, and the Emerald compiler used dataflow analysis quite extensively to avoid method lookup altogether, by compiling a direct subroutine call to the appropriate method.
Second, in those cases where dataflow analysis could not assign a unique concrete type to the target expression, we avoided the cost of searching for the correct method by inventing a data structure that took advantage of Emerald's abstract typing. This data structure was called an *AbCon*, because it mapped *Ab*stract operations to *Con*crete implementations. AbCons are the responsibility of the run-time system: it constructs an AbCon for each ⟨*type*, *implementation*⟩ pair that it encountered. An object reference consists not of a single pointer, but of a pair of pointers: a pointer to the object itself, and a pointer to the appropriate AbCon, as shown in Figure 1.
The AbCon is basically a vector containing pointers to some of the operations in the concrete representation of the object. The number and order of the operations in the vector are determined by the abstract type of the variable; operations on the object that are not in the variable's abstract type cannot be invoked, and so they do not need to be represented in the AbCon. In Figure 1, the abstract type *InputFile* supports just the two operations *Read* and *Seek*, so the vector is of size two, even though

the concrete objects assigned to f might support many more operations. An important point is that the size of the vector and the indexes to it can be determined at compile time.
For example, using Figure 1, when calling f.Seek, the compiler knows the Abstract type (*InputFile*), and can thus find the index of the *Seek* operation and generate an indirect jump via the AbCon vector. In situation (a) in Figure 1, the call would end up at *Distfile.Seek*. Doing the method lookup with an AbCon is thus reduced to a load of the AbCon vector Address, an indexing, and a load of the appropriate slot.
AbCon vectors are, in principle, created upon assignment of a variable. In the example above, the view expression returns an object reference including a pointer to an appropriate AbCon, which is dynamically created, if necessary.
It appears that we have to generate new AbCons on EVERY assignment, but in practise this is not the case. In a simple assignment between two variable of the same Abstract Type, the AbCon will be the same as both Abstract Type and Concrete Type are the same. Thus the assignment is merely copying the two pointer. This covers many assignment.
In an assignment, if the Abstract Types differ, then a new AbCon needs to be generated. However, this needs only be done once for each (Abstract Type, Concrete Type) pair. AbCons increased the cost of each assignment slightly, but made operation invocation as efficient as using a virtual function table. In practice it was almost never necessary to generate them during an assignment, because the number of different concrete types that an expression would take on was limited, often to one, or just a handful.
Many of the AbCons that the compiler can see are needed are generated at load time. If the compiler can deduce the concrete type of the assigned object, then it would merely insert a store of the address of the relevant AbCon and the AbCon address would be inserted into the code at load time. In this case, an assignment would be a copy of the object pointer and a store of a constant address.
Furthermore, in many cases the compiler could figure out, using data flow analysis, the single concrete type of an assigned object; the compiler was therefore able to generate a direct subroutine call to the concrete operation, or even in-line the operation, if it were small. In addition, if the variable used can hold reference to one single concrete type, then the entire AbCon scheme can be elided. In such a case, the variable is implemented just like a pointer variable in C and the call is as efficient as a procedure call in C.
In the case of an object of a new concrete type that arrives over the network at run time, it is necessary to generate a new AbCon dynamically, but this would typically occur in connection with the arrival of the object.

1.6 Indexing AbCons

The compiler uses the following scheme to index AbCons: First, each of the operation names is mapped to a unique id which is fetched from a shared database. Second, the operations are sorted using their unique

ids. Third, each operations is assigned a sequential index based on the sort. Thus the AbCon vector is dense and efficiently indexed by a small integer.

1.7 Single and Multiple Inheritance

Emerald does not have inheritance in the Smalltalk or C++ sense of the word. However, the AbCon mechanism is suitable for languages with traditional single or multiple inheritance: The AbCons are generated for each pair of (*interface*, *class*). Essentially, the method lookup is done at the time that the AbCon is generated rather than upon every call. Because the interface is fixed, a call can still be executed in constant time regardless of inheritance.

1.8 Caching AbCons

Each time an AbCon is to be generated, the run time system first does a double key hash lookup of the AbCon (using the unique id of the Abstract and of the Concrete Type as a double key). If the AbCon already exists, the pointer to it is returned. If it does not, a new AbCon is generated for the pair and inserted into the hash table. Thus, except for the first time an AbCon is generated, the time to find an AbCon is constant (assuming that a suitable hashing algorithm is used). Any AbCon that the compiler can see is needed, is generated at load time as the necessary is present at that time.

1.9 Performance Summary for AbCons

AbCons incur overhead as follows:
On assignment, some assignments are a double pointer load and store instead of one.
On method calls: in the worst case, the call can be performed using a load (of the AbCon vector), a load of the content of the indexed element, and a jump to it. For calls where the compiler can deduce the Concrete Type, the call is the same as a procedure call in C.
The potentially most damaging overhead is the extra storage incurred by the AbCon pointers in those variables requiring them — their storage size is doubled. This can be signficant for large arrays. However, in practice, it seems that most large arrays are not used for storing polymorphic data.

1.10 Comparison with Other Schemes

Compared to a contemporary language such as Smalltalk, method lookup in Emerald is much more efficient. Indeed, we achieved execution times comparable to similar C programs for a number of benchmarks ([6]).
Some years later, the Self project invented the Polymorphic Inline Cache [7], which is successful at eliminating message lookup for precisely the same

reason that AbCons rarely need to be generated at assignment time: the number of concrete types that an expression can take is is usually small, and after the program has run for a while, the cache always hits.

Alpern *et al* [8] give an excellent overview of previous techniques for interface dispatch. They describe an *itable* which is a virtual method table for a class, restricted to those methods that match a particular interface, i.e., essentially an AbCon, but indexed by method rather than an index. The problem is that for a given method lookup, the implementation first must lookup the appropriate itable that matches the interface in question, and thereafter search the itable for the method in question.

Alpern *et al* [8] propose a new interface dispatch mechanism, called the interface method table (IMT). They propose an IMT for each class. The IMT is essentially a hash table that contains the id of interface methods and the method's address. The IMT is populated dynamically as the virtual machine discovers that a class implements an interface; it then adds that interface's method to the class's IMT. As long as there is no conflict, the call sequence can merely load the appropriate address right out of the IMT and jump to the method. Because the IMT is a hash table, there is the possibility of conflict. Such a conflict is detected when a new entry into the IMT is made that conflicts with a previous entry. In such a case, the virtual machine generats a conflict resolution stub that picks the correct interface method and jumps to the correct method in the appropriate class.

This scheme thus tries to combine a fast lookup while reducing the size of the IMT. However, compared to Emerald's AbCons, even the shortest sequence for interface dispatch contains an extra load (of the id of the interface method called). And AbCons are dense (because the compiler knows the interface) and therefore shorter than the IMT hash tables, and there is no possibility of conflict (because it is not a hash table).

Zendra *et al* [9] describe a different approach where tables are eliminated and replaced by a simple static binary branch code using a type inference algorithm.

2 Summary

We describe the Emerald mechanism for precomputing method lookup to make method calls more efficient. The mechanism is made possible by having a strong type system requiring a type of all expressions and variables. Method calls can, in general, be performed quite efficiently (using only two memory loads) in constant, low time. The cost is that some variables need to have an extra pointer thus increasing space overhead and increasing the cost of assignment. And in cases where the compiler can determine the concerte type of an object, the compiler can elide the extra overhead and call the operation directly. This extra overhead can also be avoid in cases as the AbCon reference can be removed entirely.

Parts of this paper has appeared in earlier Emerald articles [5,10].

References

1. Raj, R.K., Tempero, E., Levy, H.M., Black, A.P., Hutchinson, N.C., Jul, E.: Emerald: a general-purpose programming language. Software — Practice and Experience **21**(1) (1991) 91–118
2. Black, A.P.: The Eden programming language. Technical Report TR 85-09–01, Department of Computer Science, University of Washington (September 1985)
3. Birtwistle, G.M., Dahl, O.J., Myhrtag, B., Nygaard, K.: Simula BEGIN. Auerbach Press, Philadelphia (1973)
4. Dahl, O.J., Myhrhaug, B., Nygaard, K.: Common base language. Technical Report S-22, Norwegian Computing Center (October 1970)
5. Black, A., Hutchinson, N., Jul, E., Levy, H., Carter, L.: Distribution and abstract data types in Emerald. IEEE Transactions on Software Engineering **SE-13**(1) (January 1987) 65–76
6. Hutchinson, N.: Emerald: An Object-Oriented Language for Distributed Programming. PhD thesis, Department of Computer Science, University of Washington (January 1987)
7. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In America, P., ed.: Proceedings ECOOP '91. Volume 512 of Lecture Notes in Computer Science., Geneva, Switzerland, Springer-Verlag (July 1991) 21–38
8. Alpern, B., Cocchi, A., Fink, S., Grove, D.: Efficient implementation of java interfaces: Invokeinterface considered harmless. SIGPLAN Not. **36**(11) (2001) 108–124
9. Zendra, O., Colnet, D., Collin, S.: Efficient dynamic dispatch without virtual function tables: the smalleiffel compiler. SIGPLAN Not. **32**(10) (1997) 125–141
10. : Hopl iii: Proceedings of the third acm sigplan conference on history of programming languages (2007) Conference Chair-Barbara Ryder and Program Chair-Brent Hailpern.

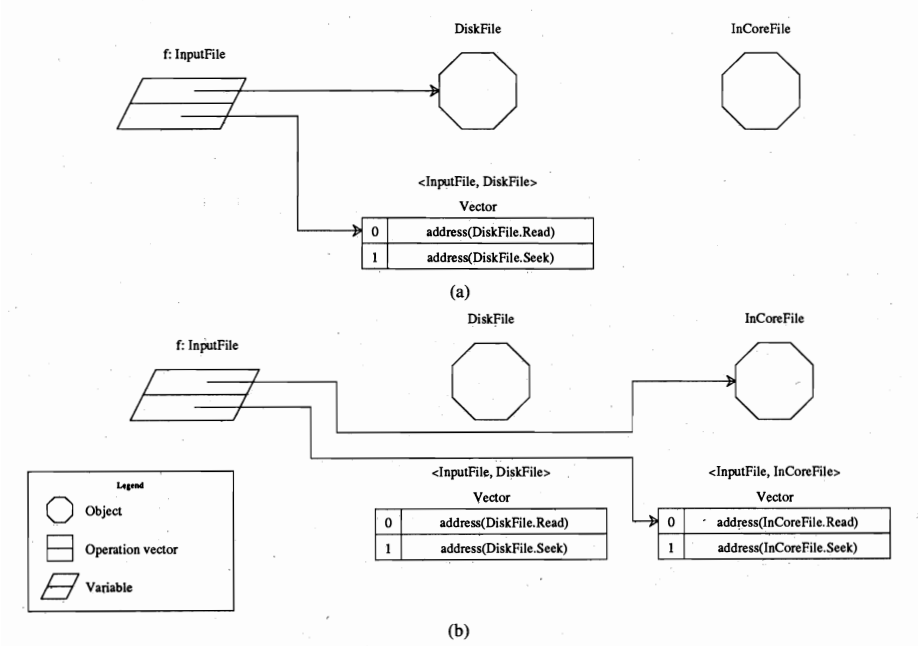

Fig. 1: This figure, taken from reference [5], shows a variable f of abstract type *InputFle*. At the top of the figure (part a), f references an object of concrete type *DiskFile*, and f's AbCon (called an Operation vector in the legend) is a two-element vector containing references to two *DiskFile* methods. At the bottom of the figure (part b), f references an object of concrete type *InCoreFile*, and f's AbCon has been changed to a two-element vector that references the correspondingly named methods of *InCoreFile*. (Figure ©1987 IEEE; reproduced by permission.)